ECE 485 Cache Simulation Project

By: Travis Hermant, Abdullah Barghouti, Alexandra Pinzon, and Ammar Khan
Instructor: Yuchen Huang

Table of Contents:

Introduction:

The purpose of this project is to design and simulate a split L1 cache for a new 32-bit processor that can be used with up to three other processors in a shared memory configuration. The system employs a MESI protocol to ensure cache coherence. It consists of an instruction cache and a data cache that employ the LRU replacement policy. The cache will keep track of the statistics such as the number of cache reads, writes, hits, misses, and the cache hit ratio.

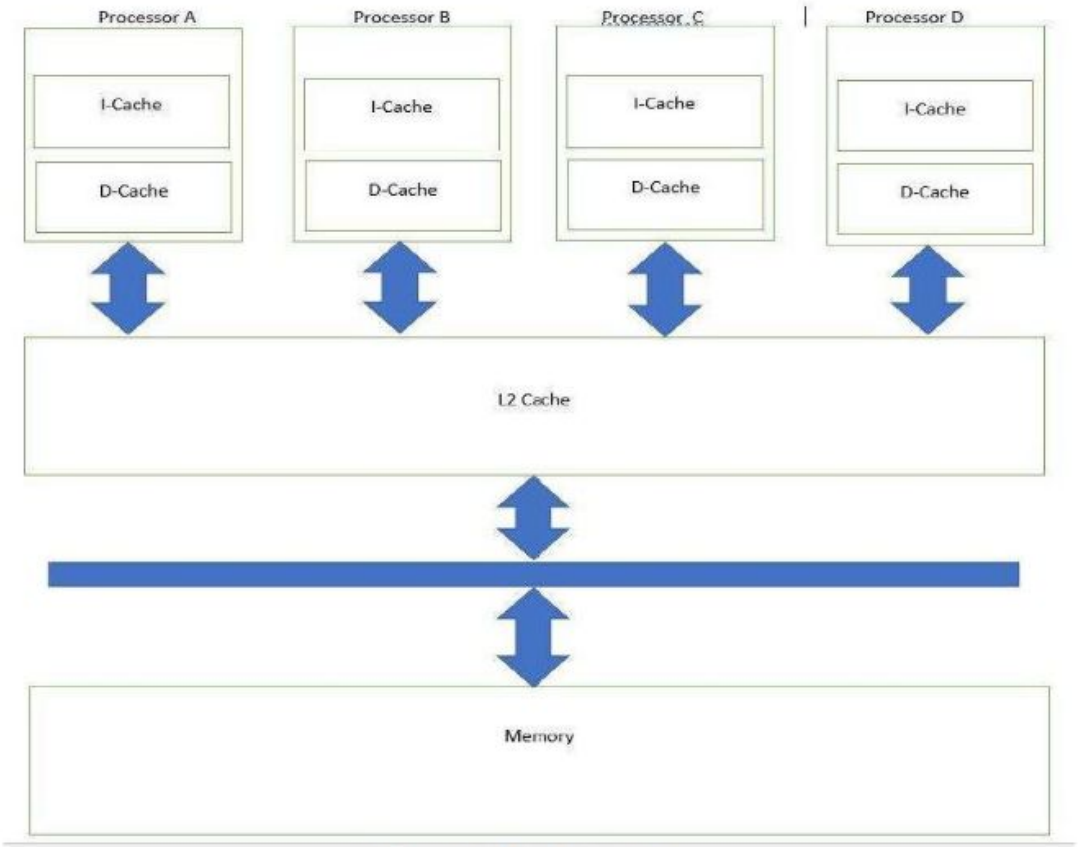
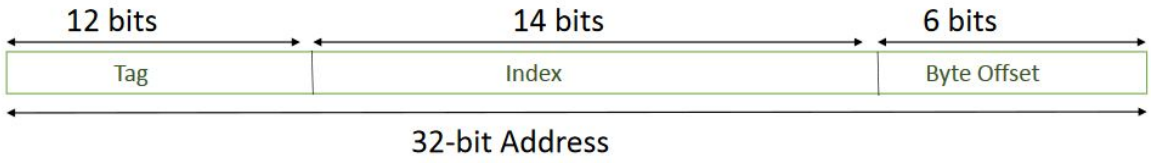


Figure 1: Block diagram

Internal Design documentation:

The L1 instruction cache is four-way set associative and consists of 16K sets of 64-byte lines, while the L1 data cache is eight-way set associative and consists of 16K sets of 64-byte lines. The L1 data cache is write-back using write allocate and write-back except for the first line, which is write-through. Both caches are backed by a shared L2 cache and employ LRU replacement policy.

The address is 32-bits, with 12 bits reserved for the Tag, 14 bits for the Index, and 6 bits for the Byte Offset. Since the split cache has the instruction and data cache with the same number of sets and lines, the address and number of bits reserved for each component remain the same for each.



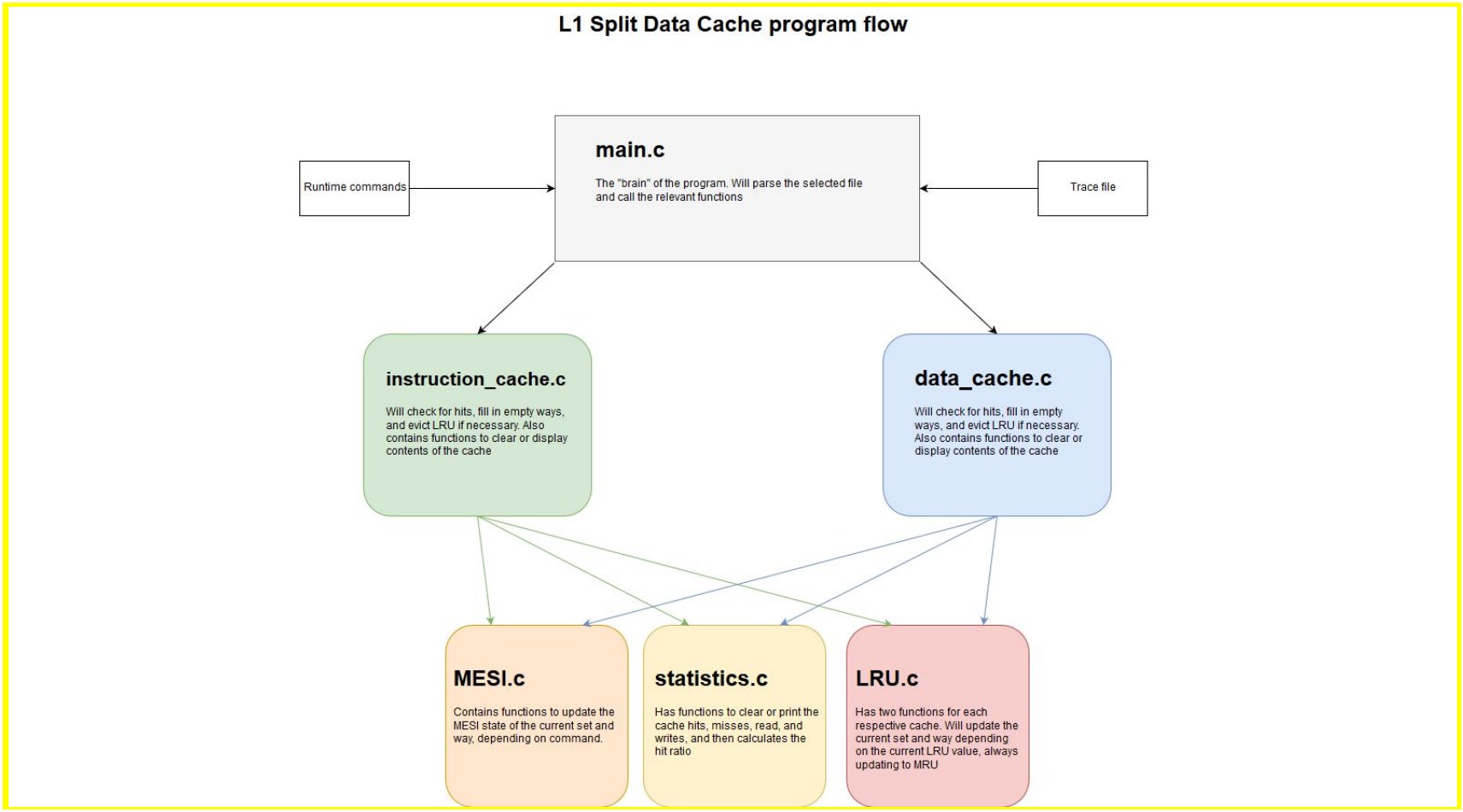
Assumptions and design decisions:

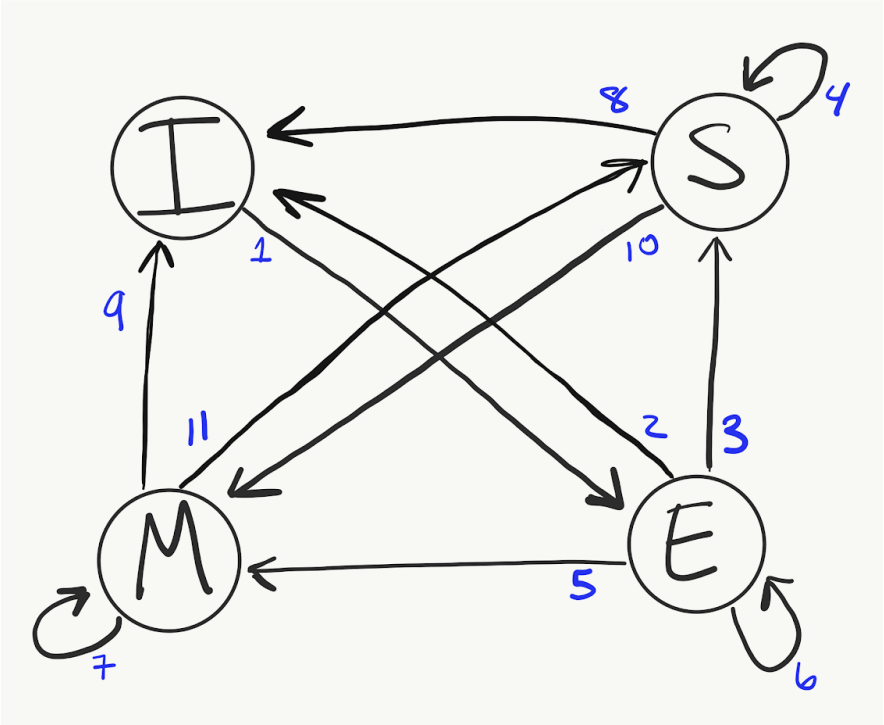
The cache was designed assuming that an L2 processor would be interacting with it at specific times, therefore creating a slightly different MESI protocol than expected. This cache design also assumes that only the very first write to the cache would be interpreted as a write-through, changing the MESI state from invalid to exclusive.

The design has LRU implemented for the instruction and data cache in a way that the MRU would be the largest number, while zero would the LRU. The structure of the cache simulation was broken up into six parts: main, data_cache, instruction_cache, statistics, LRU, and MESI, which allowed for efficient debugging. The “main” file is where all of the code was combined together and inside of the other files, depending on the trace file input, functions would be called in their appropriate places.

Data content is not a factor for this cache, instead mainly focusing on Tags, LRU, and MESI. Since there is no other literal processor that we are interacting with, a secondary processor will be assumed to exist so that every MESI state will be used. When snooping occurs in the trace file and with a fictional L2 cache in place, the snoop command will be followed by a ‘3’ command to emulate the L2 cache invalidating the data that is present.

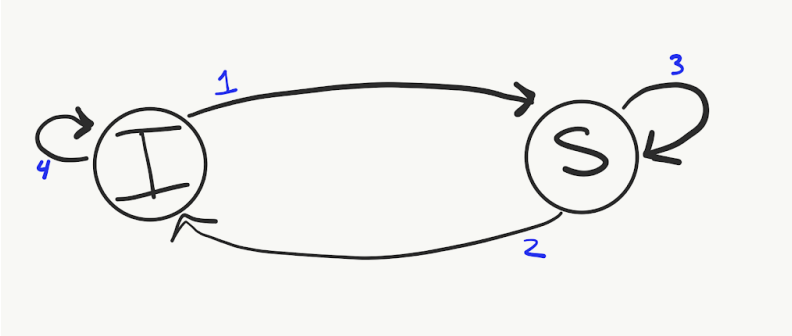
The overall behavior of the cache is described as follows: a single line is read at a time from the trace file, and parsed to separate the ‘n’ command from the address. The address will then be separated into its Tag, Index, and Byte Offset by right and left shifting according to its position in the address. The command that is read determines where the program goes next: calls to read or write data will call the data cache, reading an instruction will call the instruction cache, snooping and invalidating will skip straight to updating MESI, and the reset/print command will make a call to each of the caches to clear or print them respectively, and will either clear or print the statistics that have been saved.


L1 Split Data Cache program flow
Runtime commands
main.c
The "brain" of the program. Will parse the selected file and call the relevant functions
Trace file
instruction_cache.c
Will check for hits, fill in empty ways, and evict LRU if necessary. Also contains functions to clear or display contents of the cache
data_cache.c
Will check for hits, fill in empty ways, and evict LRU if necessary. Also contains functions to clear or display contents of the cache
MESI.c
Contains functions to update the MESI state of the current set and way, depending on command.
statistics.c
Has functions to clear or print the cache hits, misses, read, and writes, and then calculates the hit ratio
LRU.c
Has two functions for each respective cache. Will update the current set and way depending on the current LRU value, always updating to MRU



Data Cache:

1. N = 0 or N = 1, Data Read or Data write but only for first write
2. N = 3 or Reset, Invalidate command from L2
3. N = 0, Data Read
4. N = 0, Data Read
5. N = 1, Data Write
6. N = 0, Data Read
7. N = 0 or N = 1, Data read or write
8. N = 3 or Reset, Invalidate command from L2
9. Reset
10. N = 1, Data write
11. N = 4, Response to Snooping, Data request from L2



Instruction Cache:
1. N = 2, Instruction fetch
2. Reset
3. N = 2, Instruction fetch
4. Reset

Testing:

Test case for mode 0:

- Make sure cache messages are being printed

Test case for mode 1:

- Make sure the cache messages and communication with L2 is printed

N values to test for

n	**Task**
0	Read data request To L1 Data cache
1	Write data request To L1 Data cache
2	Instruction fetch (a read request To L1 Instruction cache)
3	Invalidate command from L2
4	Data request from L2 (in response to snoop)
8	Clear the cache and reset all state (and statistics)
9	Print contents and state of the cache

Running the cache through various configurations to ensure all possibilities are used:

Test cases for hit rate:

- Test for 0 hits
- Test for 1 hit
- Test for multiple hits
- Hit Ratio is correctly calculated

Test to make sure the LRU is getting updated correctly. LRU is 0, MRU is 7 or 3 for data-cache and instruction-cache respectively.

- An empty set has all LRU values of -1 (our value for an invalid LRU)
- After one command has data stored in its cache, the LRU value is 7 and the rest are -1
- After one command has an instruction stored in its cache, the LRU value is 3 and the rest are -1
- When multiple commands are input and filling the set, the MRU is of the correct value and the rest are being decremented successfully, eg the previous MRU is 1 less than the current MRU
- After the set is filled, the LRU will be evicted and every other way will have their value decremented
- If there is a hit in that set, way that is hit will become the MRU and every way with an LRU value greater than its will be decremented.
- When reset, all currently occupied sets will is have their LRU's changed to -1

Test to make sure the MESI states are updating correctly

- Write miss, so the data will be Write Through and it will be set to Exclusive
- Write hit, so the data will be Modified
- Instructions in the instruction-cache will become Shared if used, and will remain in that state unless reset/invalidated
- When reset, all currently occupied sets will become Invalid

Conclusion:

The split L1 cache we created works as intended. There was one small issue that has since been corrected where the MESI state did not reflect the cache being write-through on write-miss. However, reading and writing to either cache works flawlessly and displays the contents of the cache as they are intended to be. After completing this project, our understanding of the implementation of the LRU policy, MESI protocol, and how the data cache and instruction

cache work has greatly improved. We look forward to solidifying our understanding of computer architecture in ECE 486.

Appendix

https://github.com/travishermant/L1_Cache_Simulation

Source code

Main.h

```
void SplitAddress();

#ifndef _GENERAL
#define _GENERAL

#include	<stdio.h>
#include	<stdlib.h>
#include	<stdint.h>
#include	<math.h>
#include	<string.h>
//#include	<iostream>

#define FALSE		0
#define TRUE		1

#defineKILO		1024
#define MEGA		(KILO * KILO)

//Address information
#defineSETS		(16 * KILO)
#defineADDRESS		32
#defineLINE		64
#defineTAG		12
#defineINDEX		14
#defineBYTE_OFFSET		6
#defineINST_WAY		4
#defineDATA_WAY		8

//Trace File Format
//	Where n is:		n
#defineL1_READ_DATA	0	// read data request to L1 data cache
#defineL1_WRITE_DATA	1	// write data request to L1 data cache
#defineL1_READ_INST	2	// instruction fetch (a read request to L1 instruction cache)
#defineL2_INVALID	3	// invalidate command from L2
```

```
#defineL2_SNOOP_DATA     4        // data request from L2 (in response to snoop)
#defineRESET             8        // clear the cache and reset all state (and statistics)
#definePRINT             9        // print contents and state of the cache (allow subsequent trace
activity)

//MESI
#defineM                          0
#defineE                          1
#defineS                          2
#defineI                          3

struct cache{
        int mesi;
        int lru;
        uint32_t address;
        uint32_t tag;
        uint32_t index;
        uint32_t b_offset;
};

struct stats{
        int cache_read, cache_write, cache_hit, cache_miss;
};


#endif
```

Main.c

```
#include "main.h"
#include "statistics.h"
#include "instruction_cache.h"
#include "data_cache.h"

FILE    *fp;                            // file pointer
char     *trace_file;                // temporary buffer for trace file name
char    trace_buffer[20];         // buffer for reading lines in tracefile
char    *token;                         // Token for splitting the strings
long    buff[2];                    // Buffer for converting from string to long
int             mode;                           // 0 or 1, decides
int             n;                                      // Trace "n" commands
uint32_t address, temp_tag, temp_index, temp_offset;
int     i = 0;
int     miss = FALSE;           // miss flag for data eviction

// Initialize the caches
// Stats Cache just stores various statistics to be called and printed at the end
struct stats        Stats_Cache;
// Both Inst_Cache and Data_Cache are set to be 2D arrays, with the same number of ways (16K) and 4 ways/8 ways per set, respectively
struct cache     Inst_Cache[SETS][INST_WAY], Data_Cache[SETS][DATA_WAY];

int main(int argc, char *argv[]){
        if(argc != 3){
                printf("Mode and Trace File required \n Enter mode first and then file \n");
                printf("Mode is 0 or 1, filename format is '<file>.txt'\n");
                return -1;
        }
        else if(argc == 3){
                mode = atoi(argv[1]);
                trace_file = argv[2];
        }
        else{
                printf("ERROR");
                return -1;
        }

        // Opening the file
        fp = fopen(trace_file, "r");
```

```
        if(!fp){
                printf("Opening file failed, quitting...\n");
                return -1;
        }
        //      Initialize caches; set all MESI to I, set all LRU to -1
        InstClear();
        DataClear();

        //      Read through Trace File line by line, until there's nothing left
        while(fgets(trace_buffer, sizeof(trace_buffer), fp) != NULL){
                // Each line is a space " " separated string, so separate the string for each chunk
                i = 0;
                token = strtok(trace_buffer, " ");
    while (token != NULL){
                        buff[i] = (uint32_t) strtol(token, NULL, 16);
                        i++;
                        token = strtok(NULL, " ");
                }
                // Store the separated chunks (tokens/whatever) into global variables
                n = (int)buff[0];
                address = (uint32_t)buff[1];

                SplitAddress();

                // n refers to the command found in the tracefile, check main.h for the defines and given
description
                switch(n){
                        case L1_READ_DATA:
                                Stats_Cache.cache_read++;
                                DataRead(temp_index, temp_tag);
                                break;
                        case L1_WRITE_DATA:
                                // Call to write data, checking to see if it's held
                                Stats_Cache.cache_write++;
                                DataRead(temp_index, temp_tag);
                                break;
                        case L1_READ_INST:
                                // Call for the instruction cache to read an instruction
                                Stats_Cache.cache_read++;
                                InstRead(temp_index, temp_tag);
                                break;
                        case L2_INVALID:
```

```
                        // Occurs after a snoop, L2 sends a command to invalidate once other
processor receives data
                        UpdateMESI(temp_index, 0, n);
                        break;
                case L2_SNOOP_DATA:
                        // L2 is checking to see if L1 has other data that a different processor is
looking for
                        UpdateMESI(temp_index, 0, n);
                        break;
                case RESET:
                        // Both cache functions to reset
                        InstClear();
                        DataClear();
                        // Stat function to clear all stats
                        ClearStats();
                        break;
                case PRINT:
                        // Stat function to print
                        PrintStats();
                        // Printing the contents of all valid (MESI != I) cache entries
                        PrintInstCache();
                        PrintDataCache();
                        break;
                default:
                        // Command was not 0,1,2,3,4,8,9
                        printf("Incorrect trace %s\n", trace_buffer);
                        break;
                }
        }
        fclose(fp);
        return 1;
}

//              Address can be left and right shifted corresponding to the number of bits to isolate that
area
//              [           Address 32-bits             ]
//              [ Tag 12-bits | Index 14-bits | Byte Offset 6-bits]
void SplitAddress(){
        temp_tag = address >> 20;
        temp_index  = (address << 12) >> 18;
        temp_offset = (address << 26) >> 26;
}
```

Data_cache.h

```
#include "main.h"
#include "LRU.h"
#include "MESI.h"

// declarations for variables
//File Access
extern uint32_t address, temp_tag, temp_index, temp_offset;
extern int          mode, n, miss;
extern struct stats Stats_Cache;
extern struct cache        Data_Cache[SETS][DATA_WAY];
//Function declarations
int DataRead(int set_index, int tag_size);          //function for reading the data
int DataHit(int set_index, int tag_size);  //function that checks for data hit
int DataMiss(int set_index, int tag_size);//function that checks for data miss
void DataClear(void);   //function that clears the data cache
void DataEvictLRU(int set_index, int i);// function that evicts the LRU
void PrintDataCache();
```

Data_cache.c

```
#include "data_cache.h"

//	This function handles the flow for the data cache
//	If the data isn't hit, it will then go to see if there is an empty way to fill
//	If there's no empty way, it will then evict the LRU
int DataRead(int set_index, int new_tag){
	if(DataHit(set_index, new_tag) == FALSE){
		if(DataMiss(set_index, new_tag) == FALSE)
			DataEvictLRU(set_index, new_tag);
		return FALSE;
	}
	return TRUE;
}

// Checking if the tag is present in the set, and updating the MESI and LRU if it's needed
int DataHit(int set_index, int new_tag){
	for(int i = 0; i < DATA_WAY; i++){
		if((Data_Cache[set_index][i].mesi != I) && (Data_Cache[set_index][i].tag == new_tag)){
			miss = FALSE;
			UpdateMESI(set_index, i, n);
			DataUpdateLRU(set_index, i);
			Stats_Cache.cache_hit++;
			return TRUE;
		}
		else if (i == DATA_WAY - 1)
			return FALSE;
	}
	return TRUE;
}

//  If DataHit doesnt find it, then this function will check for an empty way and fill the data in there
int DataMiss(int set_index, int new_tag){
	Stats_Cache.cache_miss++;
	for(int i = 0; i < DATA_WAY; i++){
		if(Data_Cache[set_index][i].mesi == I){
			UpdateMESI(set_index, i, n);
			Data_Cache[set_index][i].address = address;
			Data_Cache[set_index][i].tag = new_tag;
			Data_Cache[set_index][i].index = temp_index;
			Data_Cache[set_index][i].b_offset = temp_offset;
```

```
                        DataUpdateLRU(set_index, i);
                        if(mode == 1)
                                printf( "Read from L2  <0x%08x>\n", address);
                        return TRUE;
                }
                if(i == DATA_WAY - 1)
                        return FALSE;
        }
        return TRUE;
}

//  If there are no empty ways found in DataMiss, then evict the way that is LRU
//  If data is modified, then write to L2 cache and read for ownership, otherwise just read from L2
void DataEvictLRU(int set_index, int new_tag){
        miss = TRUE;
        for(int i = 0; i < DATA_WAY; i++){
                if(Data_Cache[set_index][i].lru == 0){
                //check which state we are in
                        if(mode == 1){
                                if((Data_Cache[set_index][i].mesi == M) && (n != 1))
                                        printf("Write to L2 cache    <0x%lx>\n",
(long)Data_Cache[set_index][i].address);
                                else
                                        printf("Read from L2    <0x%lx>\n", (long)address);
                                if(n == 1)
                                        printf("Read for Ownership from L2   <0x%lx>\n", (long)address);
                        }
                        UpdateMESI(set_index, i, n);
                        Data_Cache[set_index][i].address = address;
                        Data_Cache[set_index][i].tag = new_tag;
                        Data_Cache[set_index][i].index = temp_index;
                        Data_Cache[set_index][i].b_offset = temp_offset;
                        DataUpdateLRU(set_index, i);
                        return;
                }
        }
        return;
}

/*
        Functions in the Data Cache that aren't part of normal operations
*/
```

```
// Clears the entire data cache, changes MESI to I, and LRU to -1
void DataClear(void){
	for(int index1 = 0; index1 < SETS; index1++){
		for(int index2 = 0; index2 < DATA_WAY; index2++){
			Data_Cache[index1][index2].lru = -1; //decrement LRU values
			Data_Cache[index1][index2].mesi = I;		//v
		}
	}
	return;
}

// Prints all the contents of the data cache, provided that they are not Invalid
void PrintDataCache(){
	printf("\n~~~~~~~~~~~~~~~ DATA_CACHE ~~~~~~~~~~~~~~~\n");
	for(int index_set = 0; index_set < SETS; index_set++){
		for(int index_line = 0; index_line < DATA_WAY; index_line++){
			if(Data_Cache[index_set][index_line].mesi != I){
				printf("-----\n");
				printf("SET: %lx WAY: %d MESI: %d LRU: %d ADDRESS: %lx \n",
					(long)Data_Cache[index_set][index_line].index,
					index_line + 1,
					Data_Cache[index_set][index_line].mesi,
					Data_Cache[index_set][index_line].lru,
					(long)Data_Cache[index_set][index_line].address);

			}
		}
	}
	return;
}
```

Instruction_cache.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include "main.h"
#include "LRU.h"
#include "MESI.h"

// Global variables
extern uint32_t address, temp_tag, temp_index, temp_offset;
extern int          mode, n;
extern struct cache Inst_Cache[SETS][INST_WAY];
extern struct stats Stats_Cache;

//functions
int InstHit(int set_index, int new_tag);
int InstMiss(int set_index, int new_tag);
int InstRead(int set_index, int new_tag);
void InstEvictLRU(int set_index, int new_tag);
void InstClear(void);
void PrintInstCache(void);
```

Instruction_cache.c

```
#include "instruction_cache.h"

//	This function handles the flow for the instruction cache
//	If the instruction isn't hit, it will then go to see if there is an empty way to fill
//	If there's no empty way, it will then evict the LRU
int InstRead(int set_index, int new_tag){
	if(InstHit(set_index, new_tag) == FALSE){
		if(InstMiss(set_index, new_tag) == FALSE)
			InstEvictLRU(set_index, new_tag);
		return FALSE;
	}
	return TRUE;
}

// Checking if the tag is present in the set, and updating the MESI and LRU if it's needed
int InstHit(int set_index, int new_tag){
	for(int idc = 0; idc < INST_WAY; idc++){
		if((Inst_Cache[set_index][idc].tag == new_tag) && (Inst_Cache[set_index][idc].mesi !=
I)){
				UpdateMESI(set_index, idc, n);
				InstUpdateLRU(set_index, idc);
				Stats_Cache.cache_hit++; //increment hit counter
				return TRUE;
			}
		else if (idc == INST_WAY - 1)
				return FALSE;
		}
	return TRUE;
}

//  If InstHit doesnt find it, then this function will check for an empty way and fill the instruction in there
int InstMiss(int set_index, int new_tag){
	Stats_Cache.cache_miss++; //increment miss
	for(int idc = 0; idc < INST_WAY; idc++){
		if(Inst_Cache[set_index][idc].mesi == I){
			UpdateMESI(set_index, idc, n);
			Inst_Cache[set_index][idc].tag = new_tag;
			Inst_Cache[set_index][idc].index = temp_index;
			Inst_Cache[set_index][idc].address = address;
			Inst_Cache[set_index][idc].b_offset = temp_offset;
```

```
                        InstUpdateLRU(set_index, idc);
                        if(mode == 1)
                                printf("Read from L2    <0x%lx>\n", (long)address);
                        return TRUE;
                }
                if(idc == INST_WAY -1)
                        return FALSE;
        }
        return TRUE;
}

//  If there are no empty ways found in InstMiss, then evict the way that is LRU
void InstEvictLRU(int set_index, int new_tag){
        for (int idc= 0; idc<INST_WAY; idc++){
                if (Inst_Cache[set_index][idc].lru == 0){
                        if(mode == 1)
                                printf("Read from L2    <0x%lx>\n", (long)address);
                        Inst_Cache[set_index][idc].tag = new_tag;
                        Inst_Cache[set_index][idc].index = temp_index;
                        Inst_Cache[set_index][idc].address = address;
                        Inst_Cache[set_index][idc].b_offset = temp_offset;
                        UpdateMESI(set_index, idc, n);
                        InstUpdateLRU(set_index, idc);
                        return;
                }
        }

}

/*
        Functions in the Inst Cache that aren't part of normal operations
*/

// Clears the entire instruction cache, changes MESI to I, and LRU to -1
void InstClear(void){
        for(int set_index = 0; set_index<SETS; set_index++){
                for(int idc = 0; idc<INST_WAY; idc++){
                        Inst_Cache[set_index][idc].lru = -1; //decrement lru values
                        Inst_Cache[set_index][idc].mesi = I; //invalid
                }
        }
}
```

```
// Prints all the contents of the instruction cache, provided that they are not Invalid
void PrintInstCache(void){
	printf("\n~~~~~~~~~~~~~~ INST_CACHE ~~~~~~~~~~~~~~\n");
	for(int index_set = 0; index_set < SETS; index_set++){
		for(int index_line = 0; index_line < INST_WAY; index_line++){
			if(Inst_Cache[index_set][index_line].mesi != I){
				printf("-----\n");
				printf("SET: %lx WAY: %d MESI: %d LRU: %d ADDRESS: %lx \n",
					(long)Inst_Cache[index_set][index_line].index,
					index_line + 1,
					Inst_Cache[index_set][index_line].mesi,
					Inst_Cache[index_set][index_line].lru,
					(long)Inst_Cache[index_set][index_line].address);
			}
		}
	}
	return;
}
```

Statistics.h

```
#include "main.h"

extern struct stats Stats_Cache;

void ClearStats();
void PrintStats();
```

Statistics.c

```
#include "statistics.h"

void ClearStats(){
	Stats_Cache.cache_read = 0;
	Stats_Cache.cache_write = 0;
	Stats_Cache.cache_hit = 0;
	Stats_Cache.cache_miss = 0;
return;
}

void PrintStats(){
	float ratio;
	float total;

	total = Stats_Cache.cache_hit + Stats_Cache.cache_miss;
	ratio = Stats_Cache.cache_hit / total * 100;

	printf("\n~~~~~~~~~~~~~~ STATISTICS ~~~~~~~~~~~~~~\n");
	printf("Number of cache reads: %d\n", Stats_Cache.cache_read);
	printf("Number of cache writes: %d\n", Stats_Cache.cache_write);
	printf("Number of cache hits: %d\n", Stats_Cache.cache_hit);
	printf("Number of cache misses: %d\n", Stats_Cache.cache_miss);
	printf("Hit ratio percentage: %.2f %% \n", ratio);
return;
}
```

LRU.h

```
#include "main.h"

extern struct cache     Inst_Cache[SETS][INST_WAY], Data_Cache[SETS][DATA_WAY];

//Prototypes
int InstUpdateLRU(int set ,int way);  //Use struct cache variables
int DataUpdateLRU(int set ,int way);  //Use struct cache variables
```

LRU.c

```
#include "LRU.h"

int InstUpdateLRU(int set, int way)
{
	int check = 0;
	int temp_way = 0;
	int prev_lru = 0;
	if(Inst_Cache[set][way].lru == INST_WAY - 1)
		Inst_Cache[set][way].lru = INST_WAY - 1;
	else if(Inst_Cache[set][way].lru >= 0){
		prev_lru = Inst_Cache[set][way].lru;
		for(int i = 0; i < INST_WAY; i++){
			if(Inst_Cache[set][i].lru > prev_lru)
				Inst_Cache[set][i].lru = (Inst_Cache[set][i].lru -1);
		}
		Inst_Cache[set][way].lru = INST_WAY - 1;
		check = 0;
	}
	else if(Inst_Cache[set][way].lru == -1){
	// From initialized cache, so -1 means empty way
		Inst_Cache[set][way].lru = INST_WAY - 1;
		while(way > 0){
			--way;
			Inst_Cache[set][way].lru = (Inst_Cache[set][way].lru - 1);
		}
		check = 0;
	}

return check;
}

int DataUpdateLRU(int set, int way)
{
	int check = 0;
	int temp_way = 0;
	int prev_lru = 0;
	if(Data_Cache[set][way].lru == DATA_WAY - 1)
		Data_Cache[set][way].lru = DATA_WAY - 1;
	else if(Data_Cache[set][way].lru >= 0){
		prev_lru = Data_Cache[set][way].lru;
```

```
                for(int i = 0; i < DATA_WAY; i++){
                        if(Data_Cache[set][i].lru > prev_lru)
                                Data_Cache[set][i].lru = (Data_Cache[set][i].lru -1);
                }
                Data_Cache[set][way].lru = DATA_WAY - 1;
                check = 0;
        }
        else if(Data_Cache[set][way].lru == -1){
        // so -1 means empty way
                Data_Cache[set][way].lru = DATA_WAY - 1;
                while(way > 0){
                        --way;
                        Data_Cache[set][way].lru = (Data_Cache[set][way].lru - 1);
                }
                check = 0;
        }
return check;
}
```

MESI.h

```
#include "main.h"
#include "LRU.h"

extern struct cache        Inst_Cache[SETS][INST_WAY], Data_Cache[SETS][DATA_WAY];
extern int          mode, miss;
extern uint32_t address, temp_tag;

int UpdateMESI(int set, int way, int n);
```

MESI.c

```
#include "MESI.h"

/*
L1_READ_DATA      n = 0    // read data request to L1 data cache
L1_WRITE_DATA     n = 1    // write data request to L1 data cache
L1_READ_INST      n = 2    // instruction fetch (a read request to L1 instruction cache)
L2_INVALID  n = 3    // invalidate command from L2
L2_SNOOP_DATA     n = 4    // data request from L2 (in response to snoop)
*/
int UpdateMESI(int set, int way, int n /* 'n' from trace file */){
        int check;

        if((n == 0) && (Data_Cache[set][way].mesi == I)){
        //Moving from invalid to exclusive
                Data_Cache[set][way].mesi = E;
                check = 0;
        }
        // Add transition for data eviction for miss
        else if((n == 0) && (Data_Cache[set][way].mesi == E) && (miss == TRUE)){
        //Moving from exclusive to shared, assuming other processor is reading
                Data_Cache[set][way].mesi = E;
                check = 0;
                miss = FALSE;
        }
        else if((n == 0) && (Data_Cache[set][way].mesi == E) && (miss == FALSE)){
        //If read command go to shared state
                Data_Cache[set][way].mesi = S;
                check = 0;
                miss = FALSE;
        }
        else if((n == 0 || n == 1) && (Data_Cache[set][way].mesi == M)){
        //Either read or write command, stay in modified state
                Data_Cache[set][way].mesi = M;
                check = 0;
        }
        else if((n == 0) && (Data_Cache[set][way].mesi == S)){
        //If read and in shared state stay there
                Data_Cache[set][way].mesi = S;
                check = 0;
        }
```

```
        else if((n == 1) && (Data_Cache[set][way].mesi == S)){
        //If write command and in shared state go to modified
                Data_Cache[set][way].mesi = M;
                check = 0;
        }
        else if((n == 1) && (Data_Cache[set][way].mesi == E)){
        //If write command and in the exclusive state go to modified
                Data_Cache[set][way].mesi = M;
                check = 0;
        }
        else if((n == 1) && (Data_Cache[set][way].mesi == I)){
        //If write command and in the invalid state go to exclusive
                Data_Cache[set][way].mesi = E;
                check = 0;
        }
        else if(n == 2){
                Inst_Cache[set][way].mesi =  S;
                check = 0;
        }
        else if(n == 3){
        //Invalidate command from L2
                while(way < 8){
                        if((Data_Cache[set][way].tag == temp_tag) && (Data_Cache[set][way].mesi !=
M)){
                                Data_Cache[set][way].mesi = I;
                                DataUpdateLRU(set,way);
                                return 0;
                        }
                        way++;
                }
                check = 0;
        }
        else if(n == 4){
        //Response to snooping, data request from L2
                while(way < 8){
                        if(Data_Cache[set][way].tag == temp_tag){
                                if((Data_Cache[set][way].mesi == M) && (mode == 1))
                                        printf("Return data to L2    <0x%lx>\n", (long)address);
                                Data_Cache[set][way].mesi = S;
                                DataUpdateLRU(set,way);
                                return 0;
                        }
                        way++;
```

```
                }
                check = 0;
        }
        else
                check = 1;

//Returns 0 if successfully completed
//Returns 1 if did not meet any statements, meaning error
return check;
}
```